DOCUMENT RESUME

ED 098 942 IR 001 318

AUTHOR Peelle, Howard A.
TITLE A Generalized Learning Game in APL.
PUB DATE 2 May 74
NOTE 10p.; Paper presented at the Annual Meeting of Shared Educational Computer Systems (New Paltz, New York, May 1974)

EDRS PRICE MF-$0.75 HC-$1.50 PLUS POSTAGE
DESCRIPTORS Computer Programs; *Computers; Computer Science; *Educational Games; *Game Theory; *Programing
IDENTIFIERS *APL; A Programing Language

ABSTRACT

The computer programing language APL is used to describe a "learning" game, and the functions developed are generalized to extend to a class of rules. (Author/WH)

A Generalized Learning Game in APL

Howard A. Peelle
University of Massachusetts

ABSTRACT

APL is used to describe a "learning" game, generalized to a class of rules.

INTRODUCTION

Using a programming language as a conceptual framework for describing topics has powerful implications for education--yet this approach is virtually unexplored. The rewards for using a succinct and executable notation like APL are found in simple, clear expressions which often yield insights about the underlying nature of a topic. (See references [1] - [5]) APL is particularly well-suited for describing topics involving interactive processes, such as those encountered in gaming.

In this paper, APL is used to describe a simple learning process in a computerized gaming context. A game provides a good environment for demonstrating learning by computer. Generally, games have well-defined rules and objectives, narrow enough domains of discourse, and clear criteria for evaluating performance. The meaning of "learning" is, of course, limited to the game environment and is usually defined in terms of improved performance on a specified task--such as making winning moves.

THE GAME OF "LAST-ONE-LOSES"

The game chosen here to illustrate learning by computer is "LAST-ONE-LOSES"--a variant of the ancient intellectual game of NIM. LAST-ONE-LOSES is a two-person, zero-sum (win or lose) game with complete information which involves removing objects from an initial pile of objects. The rules are simple: One player moves first and may take 1 or 2 or 3 objects from the pile; the second player moves next, likewise taking 1, 2, or 3 objects. The players alternate moves in this fashion until there is only one object remaining. Then, whoever must take the last one loses!

There exists an optimal strategy algorithm for LAST-ONE-LOSES; that is, from the start of the game, one of the players can always make moves which guarantee a win. Before continuing, the reader is invited to discover (and express!) the optimal strategy for this game.

The prior question is: How does one formally represent this game--its interactive processes, simple strategies, and mechanisms for learning?

REPRESENTING THE GAME

To begin with, the basic interactive framework for two players to compete in this game of LAST-ONE-LOSES is embodied in the program below.[1]

```
    ∇ LASTONELOSES
[1]    PRINT EGGS←?20
[2]   MAKE:MOVE←□
[3]    PRINT EGGS←EGGS-MOVE
[4]    →WIN IF EGGS=1
[5]    →LOSE IF EGGS<1
[6]   NEXT:MOVE←□
[7]    PRINT EGGS←EGGS-MOVE
[8]    →LOSE IF EGGS=1
[9]    →WIN IF EGGS<1
[10]   →MAKE
[11]  WIN:'I WON THIS GAME.'
[12]   →0
[13]  LOSE:'YOU WON THIS GAME.'
    ∇
```

Note that here the objects in the game are named EGGS.

At start of the game, the number of EGGS is random.

The messages on lines [11] and [13] are expressed from the point of view of the first player.

PRINT displays the appropriate number of goose-EGGS.

```
      ∇ PRINT EGGS
[1]     (EGGS⌈0)⍴0
      ∇
```

Here, the role of the computer is primarily that of mediator. It facilitates the playing of the game and announces the winner at the end of the game.

For the computer to participate directly in the competition as one of the players, the following editing change is needed.

```
∇LASTONELOSES[2]MAKE:□←MOVE←COMPUTER∇
```

Also, a function must be developed to generate the COMPUTER's MOVE. Writing a function which makes legal moves at random is easy:

```
      ∇ MOVE←COMPUTER
[1]     MOVE←EGGS⌊?3
      ∇
```

Note that the minimum of EGGS and ?3 ensures that the resultant MOVE is never larger than the number of EGGS available.

However, as most children will quickly articulate, this function represents an inferior strategy. It makes "dumb" moves in the most obvious situations; it does not even remember its mistakes; and, clearly, it does not improve with any consistency from game to game. A function which <u>learns</u> as it plays would certainly be more interesting.

---

[1]Please observe that this program (as well as others to follow) is designed to be as <u>simple</u> as possible--with minimal concern for efficiency but high priority on <u>readability</u>. N.B. sub-function IF used in branching statements:

```
      ∇ BRANCH←LINE IF CONDITION
[1]     BRANCH←CONDITION/LINE
      ∇
```

One trivial kind of learning involves avoiding losing sequences. Of course, the COMPUTER could be programmed to store all sequences of moves for every game it played, to search through those sequences for one which occurred before, and -- if it found a losing sequence -- to be sure not to repeat it. This is a cumbersome learning scheme, although surely effective in the long run.

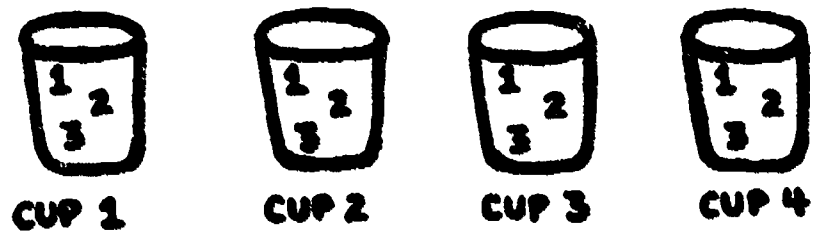
Another approach to machine learning involves building in some 'intelligence' beforehand with a structure capable of adapting itself based on 'experience.' (See reference [6] for details.) Such an "adaptive structure" provides a way of representing information about a game and utilizes rules for making changes in that information.

AN ADAPTIVE STRUCTURE

In order to build an adaptive structure for the game of LAST-ONE-LOSES, first a global variable is created: `CUPS←⍉4 3⍴⍳3` (I call the variable CUPS because one can conceptualize this matrix as four paper cups, each column containing the numbers 1 2 3.)

```
CUPS

1 1 1 1
2 2 2 2
3 3 3 3
```



Next, a function must be designed to use the CUPS structure with a rule for making a move. The rule is in two steps:

Step 1: Determine WHICH column of CUPS to use according to the current number of EGGS (as shown below):

EGGS	1	2	3	4	5	6	7	8	9	10	11	12	13	14	. . .
WHICH	1	2	3	4	1	2	3	4	1	2	3	4	1	2	. . .

Step 2: Then, PICK one of the numbers at random from that column (CUP).

This rule can be embodied in a new definition of the COMPUTER function.

```
∇ MOVE←COMPUTER
[1]   CUP←CUPS[;WHICH←DETERMINE EGGS]
[2]   →RANDUMB IF ∧/CUP=0
[3]   MOVE←MOVE←PICK (CUP≠0)/CUP
[4]   →0
[5]  RANDUMB:MOVE←EGGS⌊?3
∇
```

Note that Step 1 is expressed on line [1] and Step 2 on line [3]. The rest of the function deals with the case of a CUP with all zeros.

WHICH and MOVE record for later reference the actual numbers used to generate a MOVE.

Where sub-function DETERMINE is:

```
∇ WHICH←DETERMINE EGGS
[1]   WHICH←1+(N+1)⊥EGGS-1
∇
```

and sub-function PICK is:

```
∇ ONE←PICK MANY
[1]   ONE←MANY[?⍴MANY←,MANY]
∇
```

Now the program which accomplishes the adaptive learning can be defined. The adapting process is simply: "discard" the last MOVE which led to a loss.

```
     ∇ LEARN
[1]    CUPS[MOVE;CUP]←0
     ∇
```

This LEARN program replaces with a 0 the number in the column of CUPS WHICH represents the last (losing) MOVE.

Finally, editing the main LAST-ONE-LOSES program will allow the COMPUTER to LEARN after each game it loses:[1] ***∇LASTONELOSES[14] LEARN ∇***

LAST-ONE-LOSES now exhibits learning.

```
      LASTONELOSES
 0 0 0 0 0 0 0 0 0 0 0 0 0 0
 1
 0 0 0 0 0 0 0 0 0 0 0 0 0
□:
       3
 0 0 0 0 0 0 0 0 0
 2
 0 0 0 0 0 0 0
□:
       2
 0 0 0 0 0
 3
 0 0
□:
       1
 0
YOU WON THIS GAME.
```

```
     CUPS

 1 1 1 1
 2 2 2 2
 0 3 3 3
```

After losing the game, the adaptive structure CUPS is modified. Note that the last MOVE (3) is removed from CUP 1:

[1]To be more rigorous, i.e., to ensure that learning occurs only immediately after a game which COMPUTER lost but could have won, some additional editing changes are recommended. See Appendix for complete displays of all functions.

```
      LASTONELOSES
 0 0 0 0 0 0 0 0
 3
 0 0 0 0 0
□:
       1
 0 0 0 0
 2
 0 0
□:
       1
 0
YOU WON THIS GAME.
```

Again, CUPS has been adapted to:

```
      CUPS
1 1 1 1
2 2 2 0
0 3 3 3
```

Eventually, the adaptive structure becomes:

```
      CUPS
0 1 0 0
0 0 2 0
0 0 0 3
```

From that point on, COMPUTER will make the optimal MOVE; that is, it will win whenever a win is possible.

THE OPTIMAL STRATEGY

Some insights can be drawn from the final state of the adaptive structure. Specifically, the pattern which emerged in CUPS reveals the modular nature of the game, and hence, provides a clue to an expression for the optimal strategy. The winning *MOVE* (at any time in the game when forcing a win sequence is possible) is given by the expression *4|EGGS-1* where EGGS is the number of objects remaining. When a win is not guaranteed (the other player has a forced win), this expression yields 0. Then *MOVE* might as well be random and can be expressed as *EGGS⌊?3*

Defined as a single function, the optimal strategy is:

```
     ∇ MOVE←OPTIMAL
[1]    MOVE←4|EGGS-1
[2]    →0 IF' MOVE≠0
[3]    MOVE←EGGS⌊?3
     ∇
```

TEACHING THE COMPUTER

With the existence of an optimal strategy function, one might think of using it to TEACH the COMPUTER. What better way to demonstrate machine learning than to have one computer program train another!

After supplanting the OPTIMAL function in LAST-ONE-LOSES,

```
∇LASTONELOSES[6]NEXT:MOVE←OPTIMAL∇
```

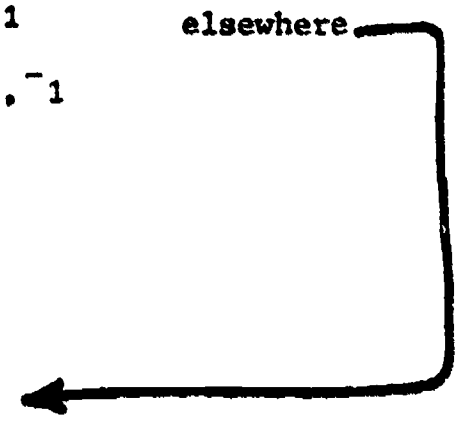
the process of learning via an adaptive structure may be automated by the following TEACH program:

```
     ∇ TEACH HOWMANY
[1]     GAMES←0
[2]   PLAY:LASTONELOSES
[3]     →PLAY IF HOWMANY>GAMES←GAMES+1
     ∇
```

One might want to revise the LAST-ONE-LOSES program to occlude unnecessary output and to record wins/losses, thusly:

```
     ∇ LASTONELOSES
[1]     EGGS←?20
[2]   MAKE:MOVE←COMPUTER
[3]     EGGS←EGGS-MOVE
[4]     →WIN IF EGGS=1
[5]     →LOSE IF EGGS<1
[6]   NEXT:MOVE←OPTIMAL
[7]     EGGS←EGGS-MOVE
[8]     →LOSE IF EGGS=1
[9]     →WIN IF EGGS<1
[10]    →MAKE
[11]  WIN:RECORD←RECORD,1
[12]    →0
[13]  LOSE:RECORD←RECORD,¯1
[14]    LEARN
     ∇
```

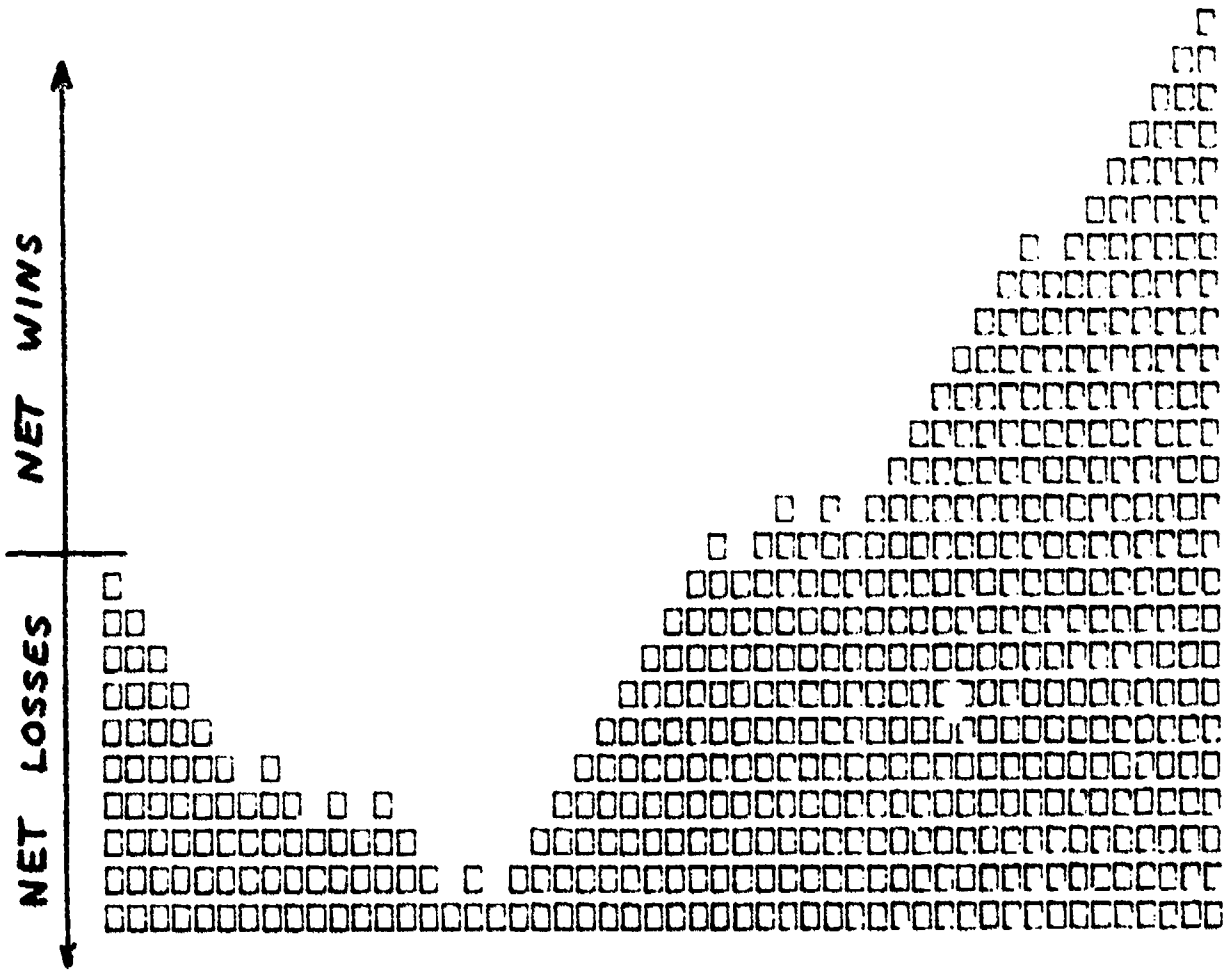
Note that the global variable RECORD must be initially specified elsewhere



```
RECORD←⍳0
```

Now, for those who wish to examine patterns of learning in more detail -- perhaps drawing analogies with biological learning -- "learning curves" can be readily produced. Based on values recorded during the TEACHing, a two-dimensional view of the COMPUTER's learning can be graphed:

```
TEACH 50
GRAPH RECORD
```



The GRAPH function used is:

```
     ∇ GRAPH RECORD
[1]    NET←+\RECORD
[2]    ADJUSTED←1+NET-⌊/NET
[3]    ' □'[1+(⍳⌈/ADJUSTED)∘.≤ADJUSTED]
     ∇
```

N.B. The +\ sum-scan function used on line [1] is not part of the IBM APL Program product.

GENERALIZABILITY

The APL functions developed so far can be generalized to extend to a class of rules. Changes in the rules of LAST-ONE-LOSES can be handled simply by substituting expressions in terms of a variable N -- where the integers ιN are allowed in any give MOVE. Specifically, the functions OPTIMAL, COMPUTER, and DETERMINE need only substitude N for 3 and N+1 for 4. (See Appendix for complete function displays.) Then, assuming that the appropriate global variables are specified beforehand,

```
N←□
CUPS←0(N+1 0)ρ⍳N
```

one is ready to play a generalized "learning" game.

CONCLUSION

This is but one topic--the topic of machine learning--which can be taught using a programming language as a conceptual framework. Many other topics are suitable for this pedagogical approach, not only topics from computer science and game theory, but also psychology, linguistics, statistics, social sciences, physical sciences, business, and ecology. Most fruitful are those topics which require explicit expression of interactive processes or models.

The challenge to educators, then, is to identify such topics and to lead students to better understandings through using A Programming Language.

REFERENCES

[1] Berry, P. et al. "APL and Insight: The Use of Programs to Represent Concepts in Teaching," IBM Tech Rpt. #320-3020, March, 1973.

[2] Iverson, K. E., "APL in Exposition," IBM Tech Rpt. #320-3010, January, 1972.

[3] Iverson, K. E., "APL as an Analytic Notation," Proceedings, APL V Conference, Toronto, May, 1973.

[4] Papert, S., "Teaching Children Thinking," MIT LOGO Memo #2, October, 1971.

[5] Peelle, H., "THE COMPUTER GLASS BOX: Teaching Children Concepts With APL," Vol. XIV, No. 4, Educational Technology, April, 1974.

[6] Block, H. D., "Learning in Some Simple Non-Biological Systems," American Scientist, March, 1965.

```
∇ LASTONELOSES;MOVE
[1]    EGGS←20
[2]   MAKE:MOVE←COMPUTER
[3]    EGGS←EGGS-MOVE
[4]    →WIN IF EGGS=1
[5]    →LOSE IF EGGS<1
[6]   NEXT:MOVE←OPTIMAL
[7]    EGGS←EGGS-MOVE
[8]    →LOSE IF EGGS=1
[9]    →WIN IF EGGS<1
[10]   →MAKE
[11]  WIN:RECORD←RECORD,1
[12]   LEARNING←OFF←0
[13]   →0
[14]  LOSE:RECORD←RECORD,¯1
[15]   LEARN
∇
```

```
∇ MOVE←COMPUTER;WHICH;CUP
[1]    CUP←,CUPS[;WHICH←DETERMINE EGGS]
[2]    →RANDUMB IF∧/CUP=0
[3]    LEARNING←ON←1
[4]    MOVE←PICK(CUP≠0)/CUP
[5]    WHICH←WHICH
[6]    MOVE←MOVE
[7]    →0
[8]   RANDUMB:MOVE←EGGS[?N
∇
```

```
∇ WHICH←DETERMINE EGGS
[1]    WHICH←1+(N+1)|EGGS-1
∇
```

```
∇ ONE←PICK MANY
[1]    ONE←MANY[?⍴MANY←,MANY]
∇
```

```
∇ MOVE←OPTIMAL
[1]    MOVE←(N+1)|EGGS-1
[2]    →0 IF MOVE≠0
[3]    MOVE←EGGS[?N
∇
```

```
∇ BRANCH←LINE IF CONDITION
[1]    BRANCH←CONDITION/LINE
∇
```

```
∇ PLAY
[1]    →0 IF LEARNING=OFF←0
[2]    CUPS[MOVE;WHICH]←0
∇
```

```
∇ SETUP
[1]    RECORD←⍳0
[2]    CUPS←⍪(N+1 0)⍴⍳N←⎕
∇
```

```
∇ GRADE RECORD;NET;ADJUSTED
[1]    NET←+\RECORD
[2]    ADJUSTED←1+NET-⌊/NET
[3]    ' ⎕'[1+(⍳⌈/ADJUSTED)∘.≤ADJUSTED]
∇
```

```
∇ EACH GAMES
[1]    PLAY:LASTONELOSES
[2]    →PLAY IF 0<GAMES←GAMES-1
∇
```